Shortest path algorithms

Single-source shortest path problem

- **Single-source shortest path problem:** given as input a weighted graph $G = (V, E)$, and a distinguished vertex s, find the shortest weighted path from s to every other vertex in G
- Input is a *weighted* graph
  - Associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the edge
- Cost of path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$. (weighted path length)
- Example: graph in 9.8 has a shortest weighted path from v_1 to v_6 with a cost of 6, going from v_1 to v_4 to v_7 to v_6

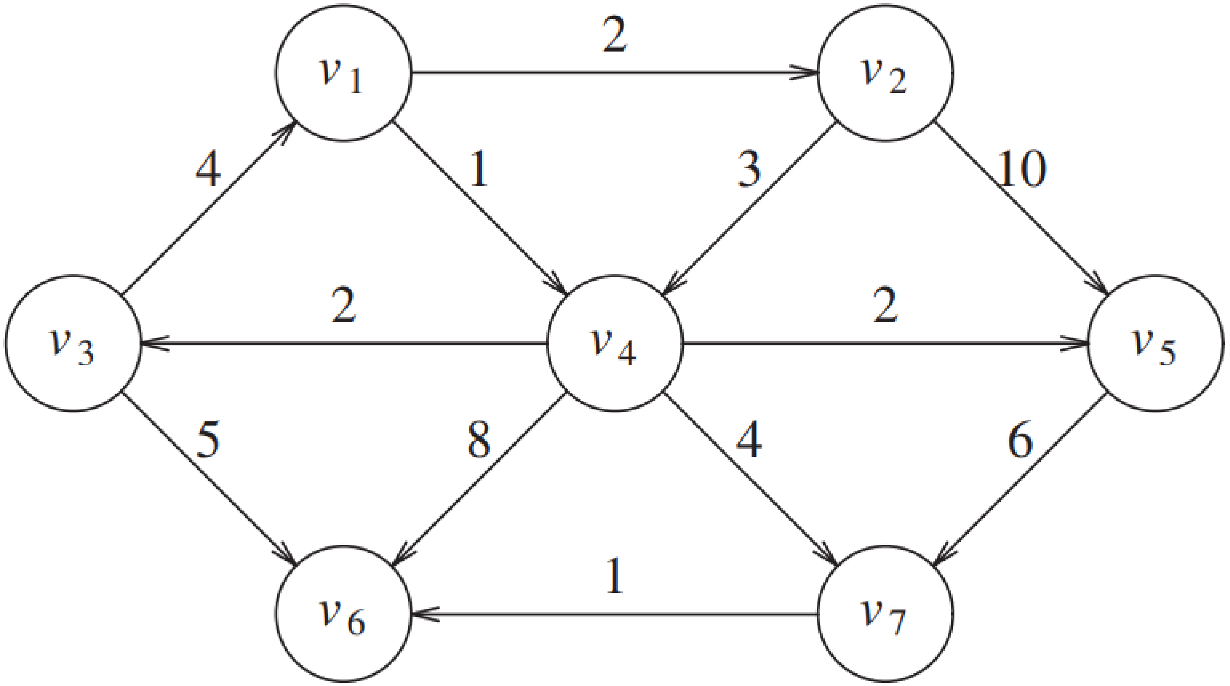


Figure 9.8 A directed graph G

Problem of negative edges

- Edges with negative weight can cause a problem when computing shortest path
- Example: edge from v_5 to v_4 has a cost 1, but a shorter path exists by following loop $v_5 v_4 v_2 v_5$, which has cost -5
- Could repeat this loop arbitrary number of times to achieve an even shorter path, thus, shortest path between these two nodes is undefined
- We call this a **negative cost cycle** and when present, shortest paths are not defined
  - Negative cost *edges* are not necessarily bad, but they make the problem harder

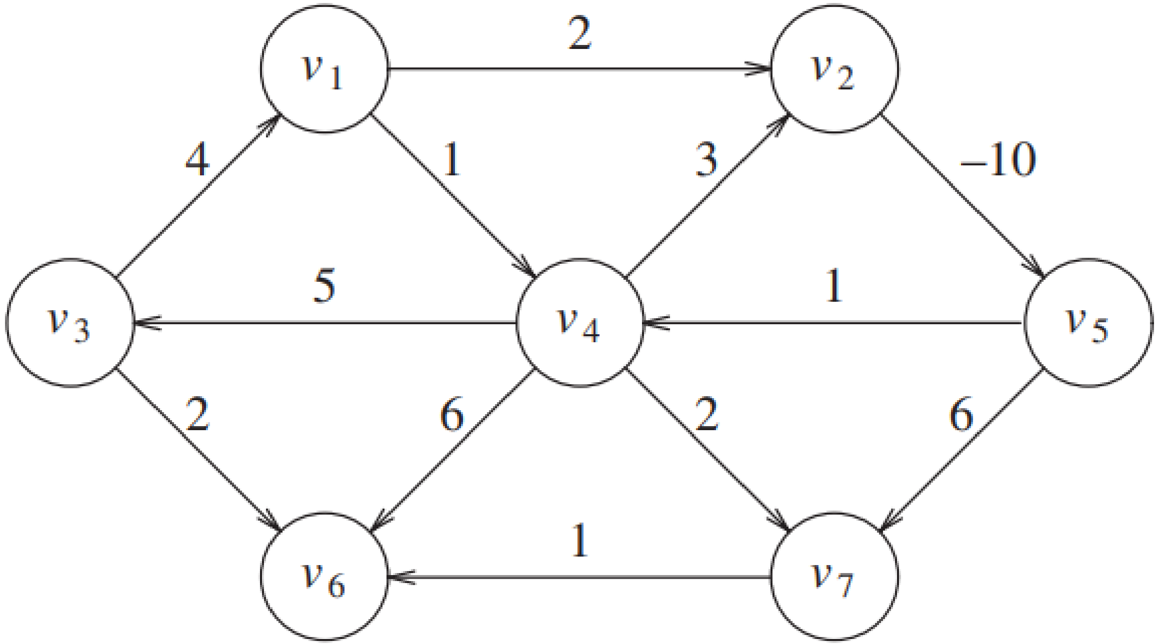


Figure 9.9 A graph with a negative-cost cycle

Unweighted shortest paths

- We will begin with a simpler problem (computing shortest path on unweighted graph)
- Using some vertex s which is given as an input parameter, want to find distance from s to all other vertices
- No weights, so we are only interested in number of edges
  - Could also assign each edge a weight of 1, making it a weighted graph

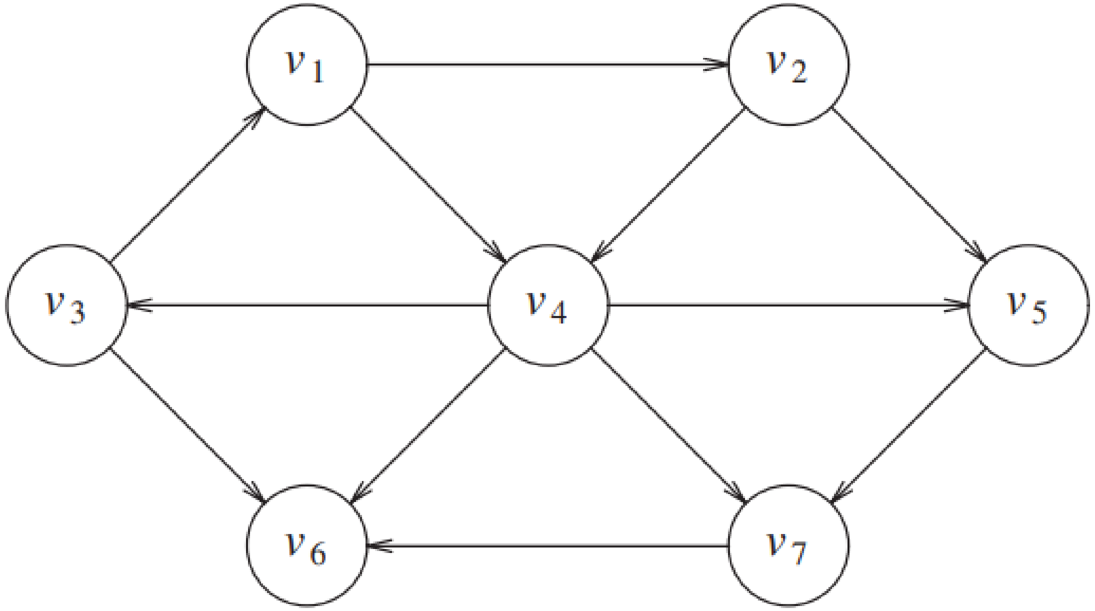


Figure 9.10 An unweighted directed graph G

Unweighted shortest paths

- Assume we are interested only in the length of the shortest paths, not the actual paths themselves (for now)
- Assume we choose s to be v_3. Shortest path from s to v_3 is 0
- Next, start looking at vertices that are distance 1 away from s. We find v_1 and v_6, and mark their distances as 1
- Next, find vertices that are distance 2 away by finding all vertices adjacent to v_1 and v_6, and whose shortest paths are not already known. We find v_2 and v_4
- Finally, by examining vertices adjacent to v_2 and v_4, we find vertices that are distance 3 away (v_5 and v_7)

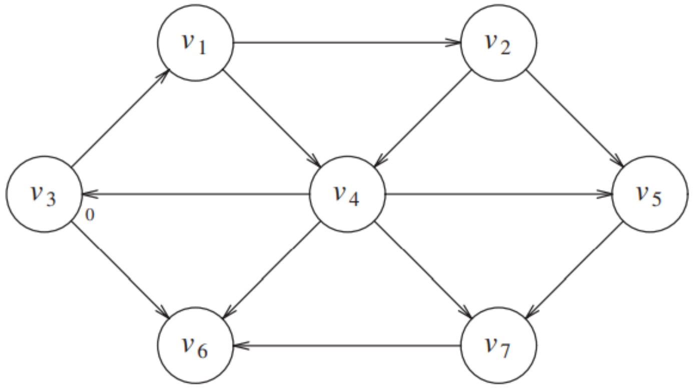


Figure 9.11 Graph after marking the start node as reachable in zero edges

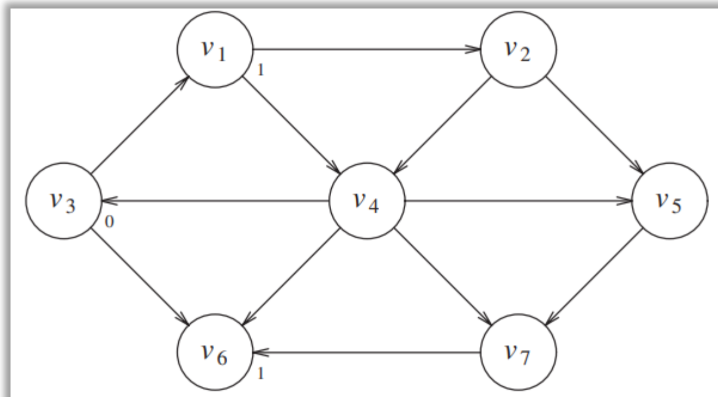


Figure 9.12 Graph after finding all vertices whose path length from s is 1

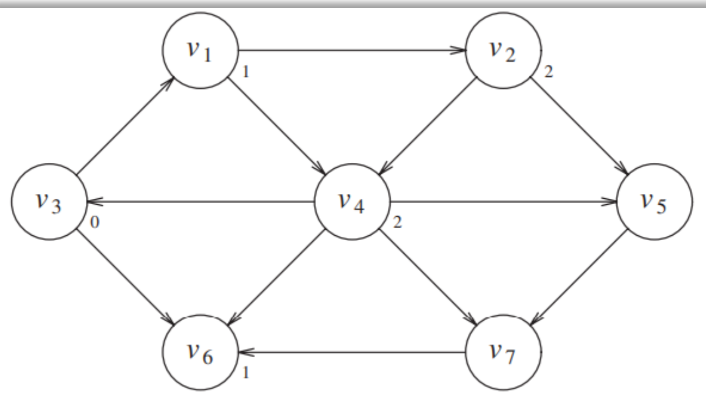


Figure 9.13 Graph after finding all vertices whose shortest path is 2

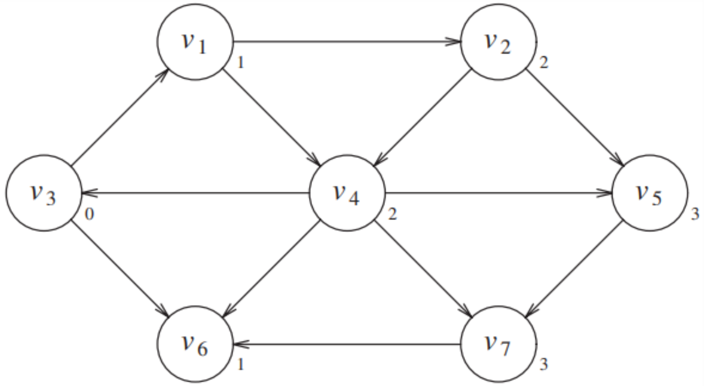
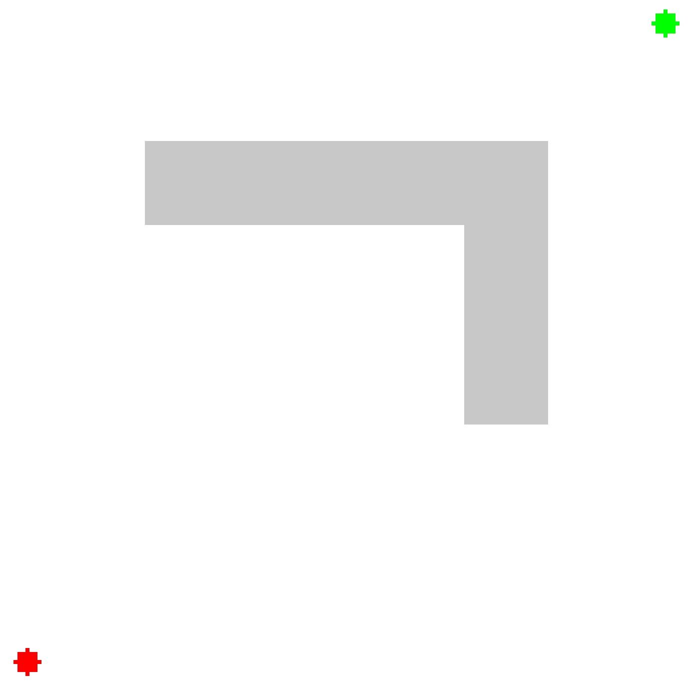


Figure 9.14 Final shortest paths

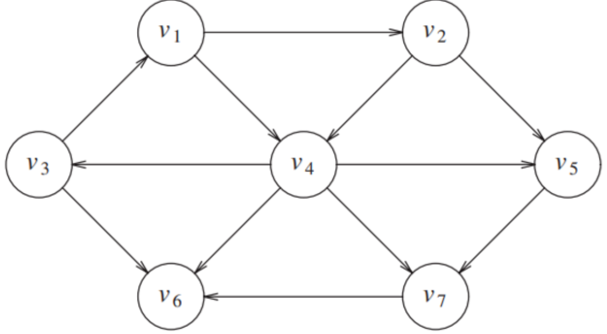
Unweighted shortest paths

- Strategy described on previous slide is a *breadth-first search*
- Operates by processing vertices in layers, vertices closest to s are evaluated first, and most distant vertices are evaluated last



Visualization of a breadth-first search

Unweighted shortest paths



- Must translate this breadth-first search strategy into code
- Will use a table to track three key pieces of information for each vertex
  - 1) distance from s in column d_v
    - Initially all vertices are unreachable, except s which has distance 0
  - 2) previous node in column p_v (will allow us to trace path)
  - 3) *known* which indicates whether or not a vertex has been processed (True or False)
    - When a vertex is marked as *known*, we have a guarantee that no cheaper path will ever be found, so that vertex is done
- Time complexity is $O(|V|^2)$ due to the doubly-nested 'for' loop

v	*known*	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Initial configuration of table used in unweighted shortest path computation

```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

Unweighted shortest paths $O(|E| + |V|)$ version

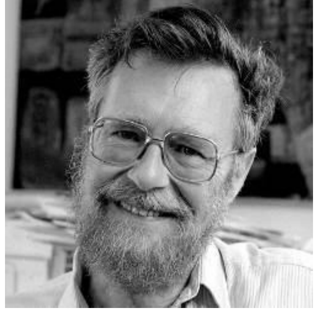
- Original version (previous slide) inefficient because outer loop continues until NUM_VERTICES-1, even if all vertices become *known* much earlier
- Instead, use a queue to hold only vertices of distance currDist
- When we add adjacent vertices of distance currDist+1, since they enqueue at the rear, we are guaranteed they will not be processed until after all vertices of distance currDist have been processed
- After last vertex at distance currDist dequeues and is processed, queue will only contain vertices of distance currDist+1
- Running time is $O(|E| + |V|)$

```
void Graph::unweighted( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

Figure 9.18 Psuedocode for unweighted shortest-path algorithm

Dijkstra’s algorithm

Edsger W. Dijkstra, 1930-2002



- When graph is weighted, the problem is (slightly) harder, but we can use the same idea as for unweighted case
- Dijkstra’s algorithm is a **greedy algorithm** for computing single-source shortest path in a weighted graph
- **Greedy algorithm:** solve a problem in stages, by doing what appears to be best at each stage
- Example: want to make change for $0.15 using *minimum number of coins*. You hold a dime ($0.10), nickel ($0.05), and 5 pennies ($0.01).
    - Greedy algorithm would simply select the largest coin (dime) and then the next largest (nickel), making change for the $0.15 in only two coins
- Greedy algorithm doesn’t always give optimal solution to every problem (for example, if we added a $0.12 coin to our collection, the greedy solution would select the $0.12 coin first, then be forced to use 3 pennies, making change in 4 coins instead of just 2
- For single-source shortest path, however, Dijkstra’s does give optimal solution

Dijkstra’s algorithm

- Proceed similarly to algorithm for unweighted case
  - Each vertex marked as *known* or *uknown,* and a tentative distance d_v is kept for each vertex, as before
  - Now, however, d_v is the shortest path from s to v using only *known* vertices
- At each stage, Dijkstra’s selects a vertex v which has smallest d_v among all *unknown* vertices, and declares shortest path from s to v as known. Remainder of stage consists of updating values of d_v

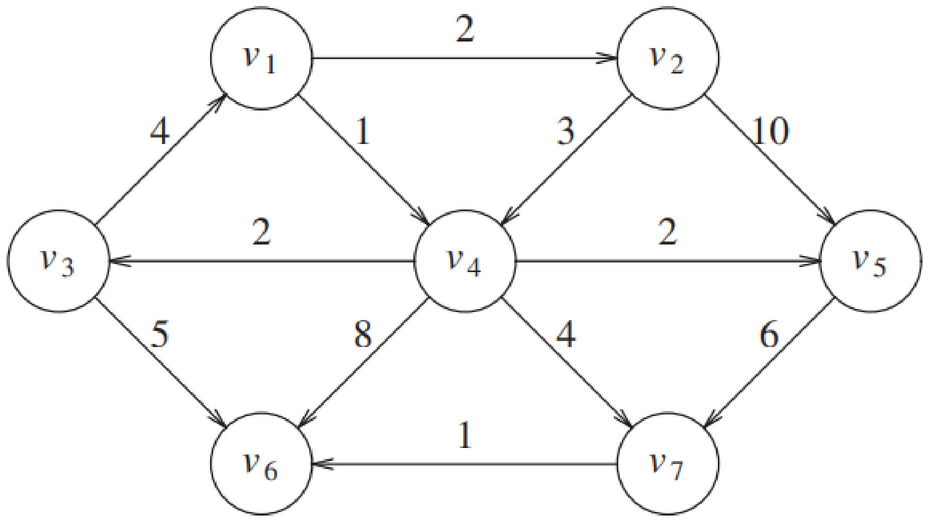


Figure 9.20 The directed graph G (again)

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.21 Initial configuration of table used in Dijkstra’s algorithm

v	known	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.21 Initial configuration of table used in Dijkstra's algorithm

v	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.22 After v_1 is declared *known*

v	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.23 After v_4 is declared *known*

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.24 After v_2 is declared *known*

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

Figure 9.25 After v_5 and then v_3 are declared *known*

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

Figure 9.26 After v_7 is declared *known*

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Figure 9.27 After v_6 is declared *known* and algorithm terminates

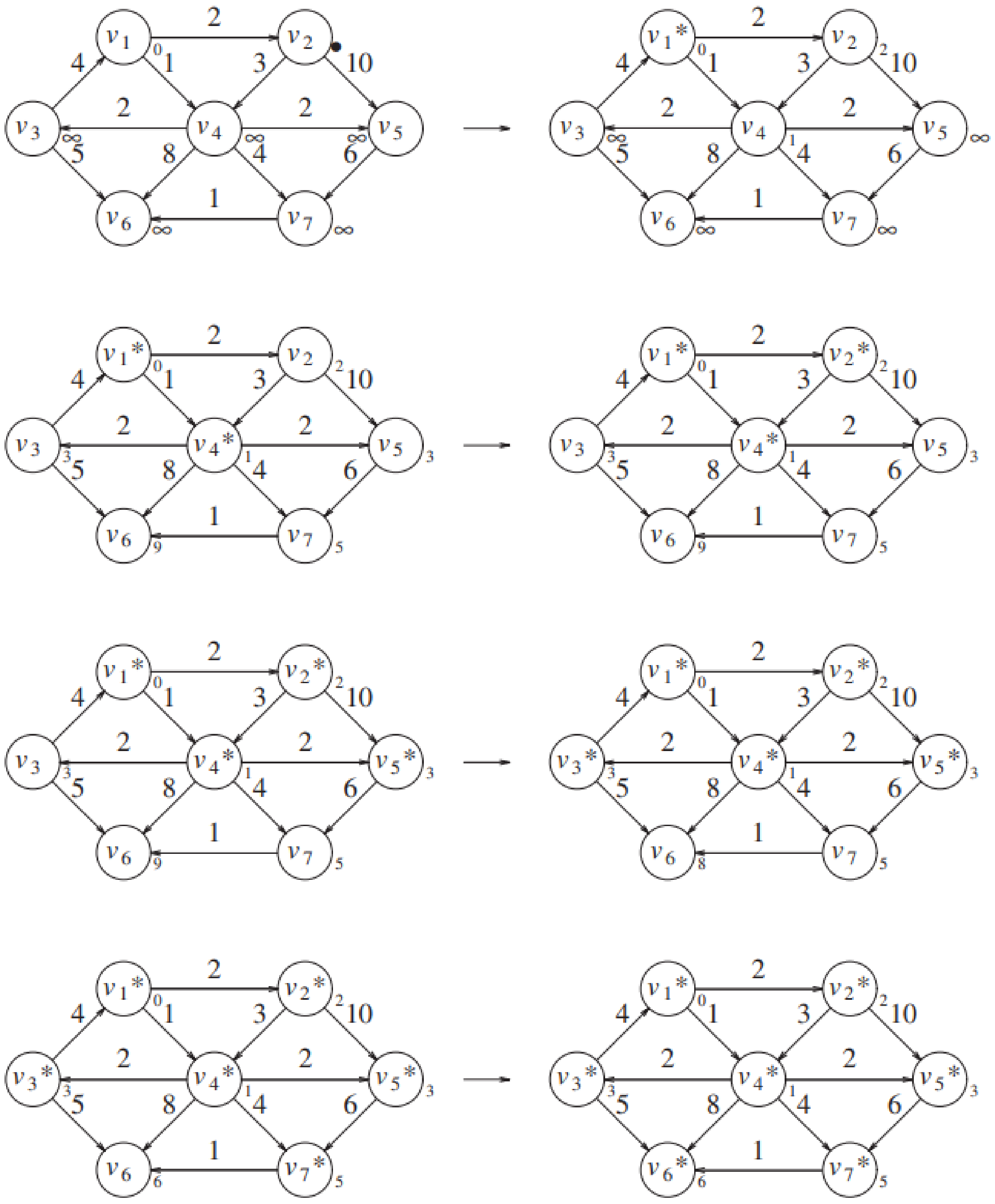


Figure 9.28 Stages of Dijkstra's algorithm

```
/**
 * PSEUDOCODE sketch of the Vertex structure.
 * In real C++, path would be of type Vertex *,
 * and many of the code fragments that we describe
 * require either a dereferencing * or use the
 * -> operator instead of the . operator.
 * Needless to say, this obscures the basic algorithmic ideas.
 */
struct Vertex
{
    List      adj;      // Adjacency list
    bool      known;
    DistType  dist;     // DistType is probably int
    Vertex    path;     // Probably Vertex *, as mentioned above
        // Other data and member functions as needed
};
```

Figure 9.29 Vertex class for Dijkstra's algorithm (pseudocode)

```
/**
 * Print shortest path to v after dijkstra has run.
 * Assume that the path exists.
 */
void Graph::printPath( Vertex v )
{
    if( v.path != NOT_A_VERTEX )
    {
        printPath( v.path );
        cout << " to ";
    }
    cout << v;
}
```

Figure 9.30 Routine to print the actual shortest path

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;

        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```

Figure 9.31 Pseudocode for Dijkstra's algorithm

Dijkstra's algorithm

- Running time depends on how vertices are manipulated
- If we sequentially scan all vertices to find minimum d_v it will take $O(|V|)$ on each scan, and hence, $O(|V|^2)$ to find minimum d_v over the entire running time of the algorithm
- Time for updating d_w is constant per update, and there is at most one update per edge for a total of $O(|E|)$
- Thus, total running time is $O(|E| + |V|^2) = \mathrm{O}(|V|^2)$
- If the graph is dense, with $|E| = \Theta(|V|^2)$, the algorithm above is asymptotically optimal (sequentially scanning vertices)
- If graph is sparse, with $|E| = \Theta(|V|)$, scanning vertices is too slow, and we should use a priority queue (heap) instead
- Selection of vertex v is a deleteMin operation, since once the unknown minimum vertex is found it is no longer unknown and must be removed from further consideration
- Update of w's distance can be done as follows:
- Treat update as a decreaseKey operation. Time to find min is then $O(\log(|V|)$, as is time to perform updates (which amount to decreaseKey operations) giving a running time of $O(|E| log|V| + |V| log|V|) = \mathrm{O}(|E| log|V|)$ (an improvement for sparse graphs)